Task Level Parallelization for Seismic Modeling and Inversion

Michel Kern, Roelof Versteeg, William W. Symes

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Task Level Parallelization for Seismic Modeling and Inversion

Michel Kern , Roelof Versteeg , William W. Symes

N° 2366

Octobre 1994

PROGRAMME 6

Calcul scientifique,

modélisation

et logiciel numérique

ISSN 0249-6399

INRIA
ROCQUENCOURT

Task Level Parallelization for Seismic Modeling and Inversion

Michel Kern , Roelof Versteeg* , William W. Symes*

Programme 6 — Calcul scientifique, modélisation et logiciel numérique
Projet Ident

Rapport de recherche n°2366 — Octobre 1994 — 13 pages

Abstract: This paper present experience with using PVM to parallelize DSO, a seismic inversion code developed in the Rice Inversion Project. We use coarse grain parallelism to dynamically distribute simulations over several workstations. When doing modeling, this strategy works efficiently, allowing us to reach an speedup of almost 4.5 on a cluster of 6 IBM RS6000 workstations. When doing inversion, however, we are currently limited to speedups of 2.4 on 3 workstations.

Key-words: Parallel Computing, Seismic Inversion

(Résumé : tsvp)

Soumis à la 3e Conférence sur les Aspects Mathématiques et Numériques des phénomènes de Propagation d'Onde, Antibes, Avril 1995

*The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892

Calcul Parallele pour la Simulation et l'Inversion sismique

Résumé : Ce rapport présente une expérience d'utilisation de PVM pour paralléliser DSO, le code d'inversion sismique développé au sein du Rice Inversion Project. Nous utilisons un parallélisme à gros grain, en distribuant les simulations sur plusieurs stations de travail. Cette stratégie se révèle efficace pour faire de la modélisation, puisqu'elle permet d'atteindre des facteurs d'accélération de 4,5 sur un réseau de 6 stations IBM RS6000. Par contre, pour une inversion complète, les accélérations sont limitées à 2,4 sur 3 stations.

Mots-clé : Calcul parallèle, Inversion sismique

1 Introduction

It has in recent years became feasible to exploit clusters of workstations as if they were a single computer, for mainly two reasons. First, individual workstations now have performance approaching 100 Mflops. Second, software to help computational scientist use a cluster is available. Using such software on a sufficiently fast workstation, it is possible to obtain performances approaching, or better than that of supercomputers. PVM, the Parallel Virtual Machine [12] is rapidly becoming the tool of choice for heterogeneous computing.

This paper shows how seismic modeling can benefit from this technique, by describing a message passing implementation of DSO, the modeling and inversion code developed in The Rice Inversion Project using PVM 3, that runs on a network of workstations. We use coarse grain parallelism, of SPMD type in which the individual tasks are full fledged simulations. Thus, even though the amount of data needed by each task is substantial, the length of the computation is such that we can reasonably expect to offset the time taken by the communications.

An outline of this paper is as follows: we first recall the main features of DSO as they pertain to the task at hand. We deduce from this view that there is a natural level of parallelism within DSO, that is independent of the physical model being simulated. Section 3 details the implementation, and section 4 shows preliminary evidence of the behavior we can expect from our program. We conclude with perspectives for further study.

2 Code structure

DSO (Differential Semblance Optimization) is a code for seismic inversion under development in the Rice Inversion Project (see [14]), that can be viewed as doing simulation, or data-fitting, of multi-experiments. DSO tries to find a model that best explains observed data, by using a variant of the least-squares technique. It assumes that these data are produced by one physical model, but come from a suite of experiments done by varying some additional parameter. The basic DSO idea is to let part of the model depend on this parameter, but to penalize the objective function in so that the optimal solution will actually be independent of the parameter. For precise statements, the reader is referred to the reference quoted above.

We give two different examples, the first one from [14] is an acoustics point-source experiment, the second one, written by J. J. Carazzone at Exxon (see [15]), describes viscoelastic plane-wave propagation.

- In the first example, the earth is described by its velocity (assumed smooth), and its reflectivity (assumed rapidly varying), and the additional parameter is the position of the seismic source. DSO lets the reflectivity depend on shot position, but since "there is only one earth", penalizes to ensure the solution does not actually depend on it. This is in some sense a "canonical" example, and most of the terminology derives from it. Thus the additional parameter is invariably called a "shot".
- For the second example, the earth is now visco-elastic but one-dimensional, the source is a plane wave and the additional parameter is the angle of incidence of the plane wave.

Again, some of the quantities are artificially allowed to depend on the angle of incidence, and this dependence is penalized.

As a consequence of this discussion, we see that the main computational task of DSO is the simulation of a large number of instances of a basic experiment, under different conditions. All these simulations are independent, and it is quite natural to process them simultaneously. For the first example above, a simulation is the solution of the wave equation with several right hand sides, each one corresponding to a different seismic source position. For the second example, we simulate the arrival of plane waves with different angles of incidence.

Simulation here has to be taken in a broad sense: as explained in [13], DSO needs not only to compute forward maps, but also their adjoints, and the associated normal maps. Nevertheless, for our present purpose, we can see all of them as black boxes, and refer to any one of them as a "simulator".

As explained in [9], our implementation of DSO is built around the principle that *generic tasks should be coded in a generic way.* A consequence is that the simulator is completely hidden from the optimization algorithm. This allows the use of the same inversion method with several different choices of propagation models.

This allows us to consider the code at a very high level: a simplified view reduces to two routines, destined to be cooperating processes (c.f. Hoare [7]).

- `simline` is the driver. In the sequential version, its role is to loop over shots, read the data for each shot, call `shot`, and write the results it receives.
- `shot` is the main computational routine. It invokes the "simulator" that does all the actual work.

Since all instances of `shot` are independent, it is a very natural idea to run them in parallel. Such an approach to parallelism has several potential advantages.

- Parallelism is at a very coarse grain. Each task is a full fledged simulation;
- It requires very little communication between the nodes;
- It can be made fault-tolerant, as the number of tasks is completely independent of the number of processors;
- It uses SPMD parallelism, thus the program can run on a distributed memory computer, but also a loosely coupled network of workstations.
- It respects the DSO framework: parallelism is still model independent.

3 Parallel Implementation

3.1 The basic algorithm

A manager–worker implementation suggests itself, with `simline` acting as the manager, or rather as a *work dispatcher*, and each incarnation of `shot` as a worker, simply treating shot

```
begin simline
  while there are still active processors do
        receive(msg) from any processor; this_worker = msg.orig
        if msg.type = result
          then
                get(task.number); write(task.results)
                if there are still shots to process
                  then
                        read(next_shot); send(next_shot) to
                        this_worker
                   else send(stop) to this_worker
                end if
           else report an error
        end if
  end do
end
```

Figure 1: Manager pseudo-code

after shot. Almasi and al. [1] have used a similar strategy to distribute a frequency domain migration, where each frequency can be computed independently. In a different domain, a very close approach has been used for the physical mapping of chromosomes [19].

This implementation has the additional advantage of providing free load-balancing. A fast worker will require (and receive) more work than a slower one, without the manager having to be concerned about balancing work between workers. One drawback is the possible bottleneck of having the manager do all of the I/O. But as we shall see on examples (section 4.2), this does not usually happen, and we even get overlap of the communication by the computation.

We show below the pseudo-code for both the manager and the worker sides. We denote the number of tasks by `ntasks`, and the number of processors by `nprocs`. The algorithm for `simline` (the manager) is shown on figure 1.

The main loop of `simline` is very simple: it waits for a message to arrive. In normal operations, this should be a worker reporting work done. The results are then stored, and if there are more tasks to process, `simline` reads the relevant data, and sends them the worker. Otherwise, simline simply sends a "stop" signal.

A feature that is not shown in figure 1 is that we implement a very simple form of read-ahead: before entering its main loop, the manager starts by sending two tasks to each worker, thus ensuring that when a worker requests a new task, one is already there waiting. Obviously, the success of this scheme hinges on the assumption that it takes longer to compute a task than to send it.

We also show the pseudo-code for `shot` the worker in figure 2.

```
begin shot
  repeat
        receive(msg) from manager
        if msg.type = new work
          then
               get(thistask.data)
               solve(thistask)
               send(thistask.result) to manager
        end if
  until (msg.type = stop)
end
```

Figure 2: Worker pseudo-code

Its structure is even simpler. After having received initial information, it waits for work to arrive from `simline`, performs the corresponding computation, and goes back to wait for the next message. Hopefully, the prefetching outlined before is efficient, so that `shot` is never blocked on the *receive* statement.

3.2 Using PVM

The previous section showed the basic algorithm we use. For a practical implementation, we need to choose a transport mechanism, in other words to specify how different processors communicate. In [1], the authors use Remote Procedure Calls, and in [19], the UNIX mail was used. This is not suitable for our purposes because of the large amount of data to be exchanged. We have chosen PVM as our message passing mechanism, because of its wide availability, and also because it is rapidly becoming a *de facto* standard for distributed applications.

PVM stands for Parallel Virtual Machine. Its allows a heterogeneous network of computers to appear as one single distributed memory computer. PVM is composed of a daemon than enables PVM to run on a given machine, and of a user-callable library of routines that allow the user to start tasks on remote machines, send and receive messages between different processors, and provides tools to manage these different processes. We refer to [5] and to the User's Guide [6] for further details concerning the package.

Let us point out two features of PVM of which we made particular use.

- Messages are typed. This greatly increases program clarity. It allows for instance the manager to decide whether the message just received is a result form a worker, or an error message from a worker.
- All messages exchanged by PVM have to be packed. This has the double advantage of allowing different data types in one message, and mostly of reducing the overall number of messages exchanged. A minor inconvenience is that the receiving process needs to unpack the data in exactly the order in which it was packed by the sending process.

3.3 Instrumentation

Assessing the performance of *any* program is difficult, and programmers are notoriously poor at guessing where to optimize a code. This is even more true of parallel programs, where it is compounded by the necessity to take into account not only the computations but also the communication delays. Tools are essential in order to achieve an understanding of the kind of performance we are getting out of a code, and of the bottlenecks.

An essential aspect of parallel programs is their dynamic character. Accordingly, a visual display of the time evolution of the program is important. This requires the code to be instrumented, then *log-files* are created while the program runs (hopefully with a minimum impact on the behavior of the program), and the log-files are interpreted graphically a-posteriori.

The tools we used are two packages working together: `alog` is used to instrument the code. It allows the user to define *events*, to be logged to a file. The second package, `upshot`, interprets the log-files in a graphical way using the X Window System. They are both described in [11].

The results presented in section 4 were obtained using the tools described here.

4 Experimental results

4.1 Modeling example

The examples in this section are described more fully in [8]. Other examples, based on the visco-elastic model of [15] are described in [10].

4.1.1 Marmousi

We run here an adjoint map on a scaled-down version of the so-called "Marmousi" model [3]. We used 24 shots, on a 1650×450 grid, with 1450 time steps. This experiment was performed at the IBM Center in Dallas, on a cluster of 8 RS600 370 workstations, of which one acted as the manager (no worker ran on the manager node). The time and Mflops rate on a single node were 624 s and 40 Mflops. The time with 6 workers is 45 mn 24 s, giving a speedup of 4.4 (or an efficiency of 72 %). It is interesting to note that this particular example took 36 mn on a Cray Y-MP, running at 280 Mflops.

4.1.2 The gas cloud

The geophysical implications of this example are described in [17], and is consists of a low velocity perturbation in an otherwise layered medium. The experiment consists of simulating 30 shot positions, on a 512×128 grid. There are 500 time steps, and 34 receiver positions. This is representative of the model sizes we are currently able to use for inversion, as shown in the next subsection.

We first ran on a single node to evaluate floating performance of the chip. The execution time was 527 s, the elapsed time was 560 s, or 94% use of the machine. On a Cray Y-MP (at Cray Research Corporate Computing Network), execution time was 83 s. The Mflops rate were 254 on the Y-MP, and 40 Mflops on the RS6000.

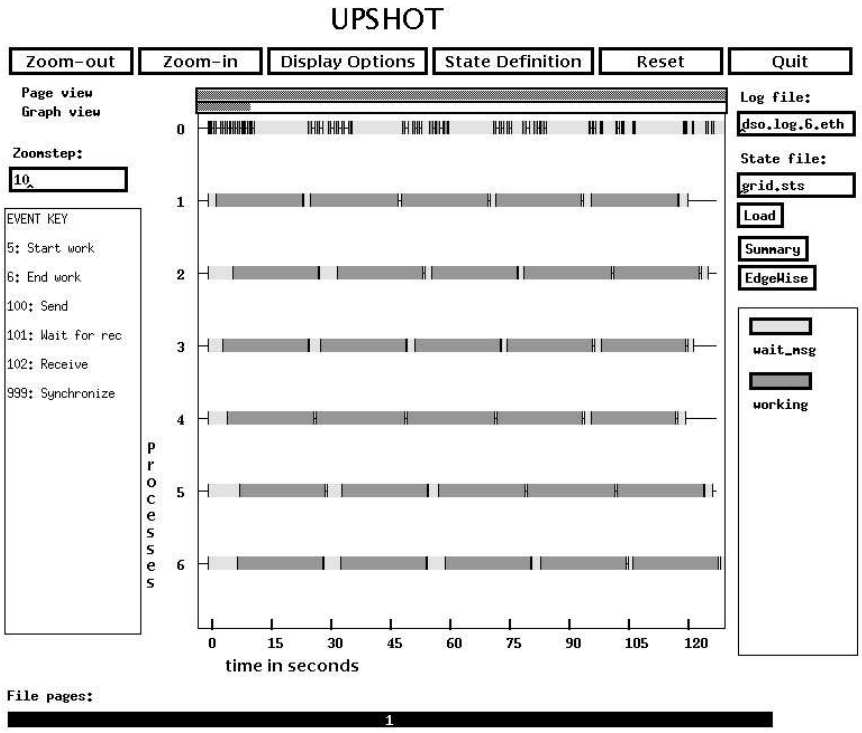

Figure 3: Upshot output for example in section 4.1.2

Then, we ran on a the cluster, using 6 workers in addition to the manager. Execution time was 130 s. We also used `alog` and `upshot` to produce and examine a log file for this run. It is shown on figure 3. The graph shows a processor versus time graph of the execution. Each horizontal line is a processor, with line 0 being the manager. The darker portions of the line are actual computing periods, while the lighter parts are idle time, when the processor is blocked waiting for a message.

As is apparent on the figure, the workers are busy computing most of the time. Also, the master has only bursts of activity, when it is doing I/O. This is the expected behavior, and justifies our design decisions, at least for doing simulations. The next section will somewhat weaken our enthusiasm.

4.2 Inversion example

The example discussed here is taken from [18]. The physical model is the same as in 4.1.2, but we are now attempting inversion. This includes a significant amount of linear algebra which is not parallelized. Note that a run in this section is the equivalent of 58 runs of the previous section.

We ran on a dedicated cluster of 4 IBM RS600 workstations, connected by a 10 Mb/s Ethernet. Performance results are shown in table1 (maximum speedup is computed from Amdahl's law, see 4.3, and actual speedup is what was actually measured).

Configuration	Serial	2 workers	3 workers
Time (s)	33600	19380	13980
Mflops	43	75	104
Max. speedup	1	1.76	2.38
Actual Speedup	1	1.73	2.4

Table 1: Performance for inversion example

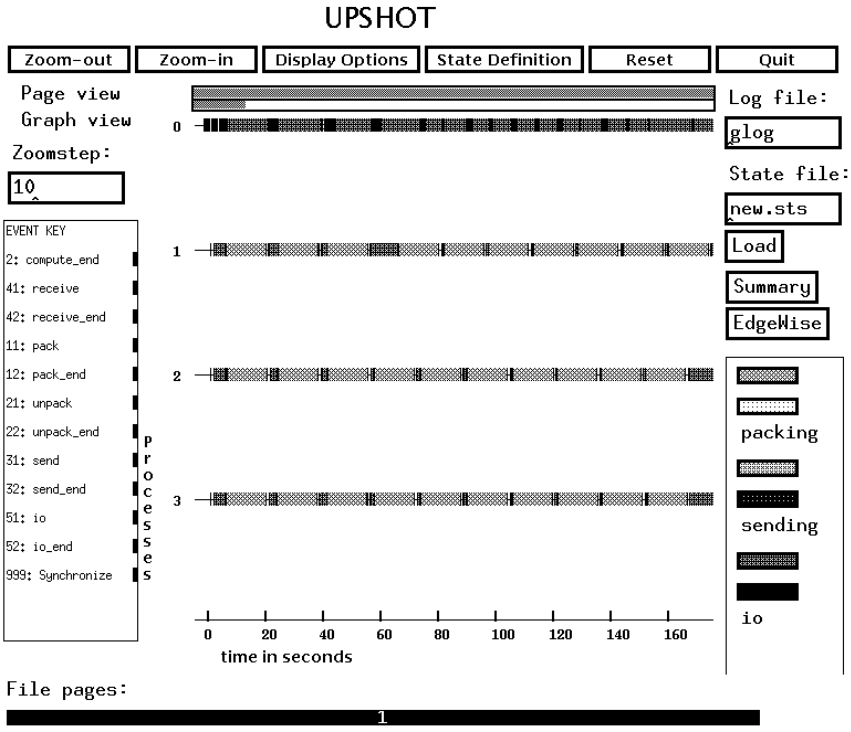

Figure 4: Upshot output for example in section 4.2

So have we violated Amdahl's law? Obviously not. We have benefited from super-linear speedup due to computation and I/O overlap, which is not taken into account by Amdahl's law.

In this case, again, our workers were kept busy all the time, their loads were balanced, and communication costs were insignificant, as shown on figure 4.

4.3 Discussion

We cannot be completely satisfied with the results just shown. We computed from the serial the parallel and serial fractions of the code. The parallel fraction (the time spent in the model dependent side) is 87%. Now Amdahl's law [2] lets us predict what the speedup will be on any given number of processors, knowing the parallel fraction of the code, via the formula:

$$S = \frac{p}{f + p(1 - f)} \tag{1}$$

where f is the parallel fraction of the code and p is the number of processors. The real content of this formula is that the speedup is limited by the *serial* part of the code. This is exemplified in figure5 which shows the speedup obtained versus the number of processors for a parallel fraction of 90%, 95%, and 99%.

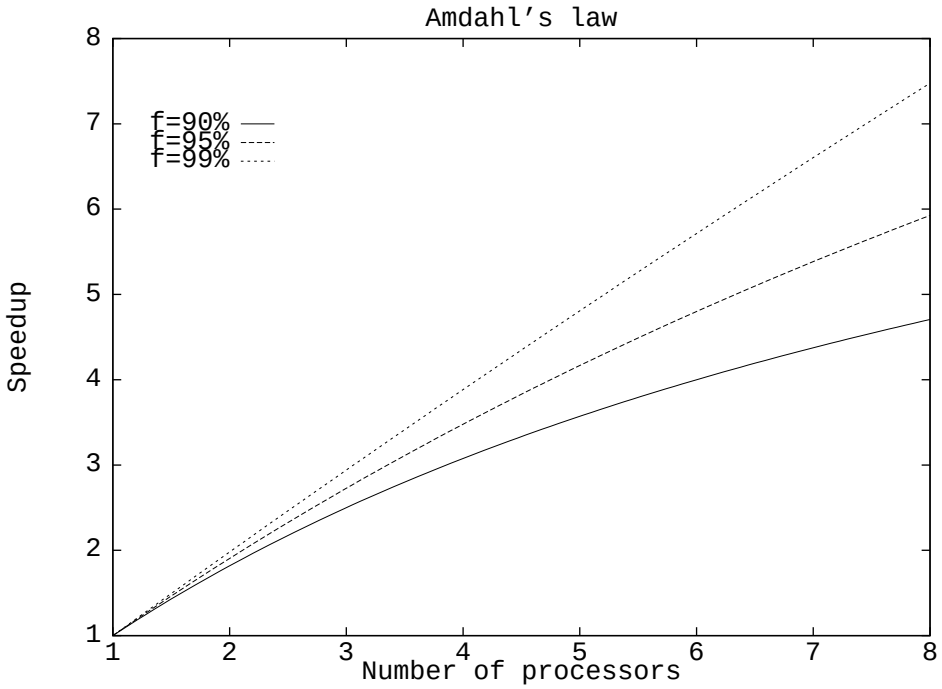

Figure 5: Amdahl's law

In our case, this means that we will never go beyond 8, whatever the number of processors, and on 8 processors, the speedup will be 4.2. This did not happen when we did modeling, where we got a speedup of almost 5 on 6 workers.

The difference is that a "simple" adjoint computation, as we did in section 4.1 has much less serial computations than the full non-linear inversion of the previous section. This comes

from two sources: The non-linear inversion algorithm has to compute several inner-products, and this is not parallelized. Also, computing the value of the objective function involves combining data to get one number, and this is a bottleneck. The linear algebra could easily be parallelized, but the computation to communication ratio is quite small, and it is not clear this would lead to any improvement. As to the serial bottleneck, there is not much that could be done without switching to a parallel optimization method (see for example [4] for an non-derivative example).

For a more detailed discussion of this example see[18].

5 Conclusion and Perspectives

We have presented a task level parallel implementation of DSO. We have described the implementation, given the rationale on which it is built, and we have presented some preliminary evidence that the design works, at least for modeling. We have also shown its limitations for doing inversion.

Obviously, this is only the beginning if this study. We wish to conclude this paper by listing several points that we could not explore but that are of immediate interest:

- We are currently missing a theoretical understanding of the behavior of the program. As mentioned in section 4, we have some idea of the general behavior of the algorithm when the number of processors increases, but we need to develop a sounder foundation. It seems that the analysis in the appendix of [1] should be applicable to our problem, but this requires more analysis.
- We plan to apply our method to Kirchoff modeling (see [16]). This cannot be done blindly, as there is more interdependence between tasks. For instance, computations share travel-time tables. This will need to be either recomputed, which is wasteful, or sent to all processors. Also, Versteeg has shown that it is more efficient to compute several shots at once, so as to amortize the reading of these travel times. Thus, we need to modify our basic framework to accommodate these changes. We hope to be able to report on this application in the near future.

Acknowledgments

This work was partially supported by the National Science Foundation, the Office of Naval Research, the Texas Geophysical Parallel Computation Project, the Schlumberger Foundation, and The Rice Inversion Project. TRIP Sponsors for 1994 are Advance Geophysical, Amoco Production Co., Conoco Inc., Cray Research Inc., Exxon Production Research Co., Interactive Network Technologies, Mobil Research and Development Corp., and Texaco Inc.

The authors thank Michael Pearlman and Virginia Torczon for helpful discussions.

References

[1] G.S. Almasi, T. McLuckie, J. Bell, A. Gordon, and D. Hale. Parallel distributed seismic migration. *Concurrency Practice nd Experience*, 5(2):105–131, April 1993.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing. *AFIPS Proc. of the SJCC*, 31:483–485, 1967.

[3] Aline Bourgeois, Patrick Lailly, and Roelof Versteeg. The Marmousi model. In R. Versteeg and G. Grau, editors, *Practical Aspects of Inversion: the Marmousi Experience*, The Hague, 1991. EAEG.

[4] J. E. Dennis, Jr. and Virginia Torczon. Direct search methods on parallel machines. *SIAM J. Optimization*, 1(4):448–474, November 1991.

[5] Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sundaram. Integrated pvm framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, 1993.

[6] Al Geist, Adama Beguelin, Jack Dongarra, Robert Manchek, and Vaidy Sunderam. PVM 3.0 user's guide and reference manual. Technical Report TM-12187, Oak Ridge National Laboratory, February 1993.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[8] M. Kern. Using DSO to benchmark the IBM SP-1 cluster. Technical report, The Rice Inversion Project, Rice University, 1993.

[9] Michel Kern and William W. Symes. Loop level parallelization of a seismic inversion code. Technical report, The Rice Inversion Project, 1993. Annual Report.

[10] Michel Kern and William W. Symes. Task level parallelization of DSO. Technical report, The Rice Inversion Project, Rice University, 1993. Annual report.

[11] Ewing Lusk. Performance visualization for parallel programs. *Theoretica Chimica Acta*, 84:377–384, 1993.

[12] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experience and trends. *Parallel Computing*, 20(4):531–545, 1994.

[13] W. W. Symes. DSO user's manual. Technical report, The Rice Inversion Project, Rice University, 1992.

[14] W. W. Symes. A differential semblance criterion for inversion of multioffset seismic reflection data. *J. Geoph. Res.*, 98:2061–2073, 1993.

[15] W. W. Symes and J. J. Carazzone. Velocity inversion by differential semblance optimization. *Geophysics*, 56(5):654–663, 1991.

[16] W. W. Symes and R. J. Versteeg. Kirchhoff modeling, migration and inversion: Theory and implementation. Technical report, The Rice Inversion Project, Rice University, 1993. Annual Report.

[17] William W. Symes. DSO velocity inversion: a gas cloud synthetic example. Technical report, The Rice Inversion Project, Rice University, 1993. Submitted to Geophysics.

[18] R. J. Versteeg, M. Gockenbach, M. Kern, and W. W. Symes. Task level parallelization of a seismic inversion code using PVM. Tulsa, 1994. SEG.

[19] Steven W. White and David C. Torney. Use of a workstation cluster for the physical mapping of chromosomes. *SIAM News*, 26(2), March 1993.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249- 6399